DISPLAYING AND IMPROVING RUN-LENGTH ENCODED IMAGES ON A BTOS SYSTEM

by

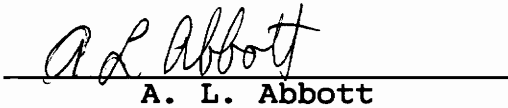
Patrick H. Geddes

Master of Science

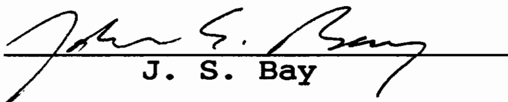
in

Electrical Engineering

F. J. Ricci, chairman



A. L. Abbott



J. S. Bay

December, 1992

Blacksburg, Virginia

LD
5655
V851
1992
G433
c.2

c.2

ABSTRACT

The goal of this project was decode and display a run-length encoded image, and to apply error concealment techniques to the decoded image. The software was written in the C programming language, a language learned during the course of this project. This software was written to read a run-length encoded ASCII file and translate it to a bit-mapped ASCII file. The software can display the bit-mapped image on a video screen and print it on a laser printer. The software also implements four error concealment techniques and displays the improved image on the video screen and on a laser printer.

This software was written for the U.S. Coast Guard's Standard Workstation, running the Unisys Corporation's Burroughs Technology Operating System (BTOS), using the BTOS C compiler. Most of the code should be transportable to other operating systems, but the display functions make use of system library functions, which may not be available on other operating systems.

Table of Contents

		Page
1.0	Introduction	1
1.1	Background	2
2.0	Previous work	4
3.0	Display code	5
3.1	Reading a run-length file	5
3.2	Bit-mapped file display	8
4.0	Error detection and concealment	16
4.1	Error detection	17
4.2	Error concealment	18
5.0	BTOS interfaces	24
6.0	Project accomplishments	26
7.0	Future work	27
References		28
Appendix A	Program listing	29
Vita		45

List of Figures

Figure		Page
1	Run length encoded data	7
2	Bit-mapped data	12
3	32x32 pixel output from Figure 2 data	13
4	64x64 pixel output	14
5	128x128 pixel output	15
6	PW error concealment	20
7	PPL error concealment	21
8	PLW error concealment	22
9	NDPL error concealment	23

1.0 Introduction

The field of digital image processing has grown rapidly in the last few decades to the point where it now has a significant impact on our daily lives. In the early 1920's image processing techniques were used to transmit pictures across the Atlantic in less than three hours [1]. It wasn't until the mid-1960's, however, that computing technology began to develop to the point where the amount of information in a typical photograph could be transmitted, stored, and processed quickly. The ever increasing processing power available on desktops has allowed complex algorithms to be applied to digitized images, and, to a certain extent, to make up for degradation of the image which occurred during its journey from one location to another. Image encoding and compression techniques applied using this technology have further contributed to the amount of information which can be quickly transmitted and efficiently stored.

Digital facsimile (fax) is one example of the impact of this field on our daily lives. Fax technology scans a document line by line and creates a run-length encoded image file. This file is transmitted to a receiving fax machine, which may be a desktop computer. Documents as diverse as copies of official correspondence and pictures to lunch orders and requests to your favorite radio station may be sent across town or across the ocean in a matter of minutes.

The goal of this project was to use the power of a desktop computer to take a computer file containing information in a compressed, run-length encoded format, expand it to a bit-mapped format, display the image on a video screen, and print it on a laser printer. Error concealment techniques would then be applied to improve the image, display the new image on a video screen, and print it on a laser printer.

1.1 Background

The ideas behind the digital fax machine provided the basic approach for this project. In digital facsimile, a document is encoded one very thin row at a time. The row is divided into a series of picture elements called pixels. In most fax machines each pixel will be scanned as either black or white and coded as one or zero. The scanned row of ones and zeroes is then compressed by run-length encoding. In run-length encoding, the number of consecutive pixels of the same color is counted and assigned a code word to represent that "run". Group 3 fax machines, which include a large majority of the machines curently in use, use modified Huffman run-length code [2]. This coding scheme assigns a code word to each number of consecutive black pixels in a row, and each number of consecutive white pixels in a row. It also assigns a code word to the end of each row, to assist in error detection. These code words are of varying

length, with the shortest code words being assigned to run lengths which occur most often, based on empirical data. In most cases, the number of bits in the binary code word representing a run is less than the total number of bits in the run. This project uses a simpler version of run-length encoding which, while not as efficient as the Huffman code, still provides a measure of data compression. Transmitting the compressed file takes less time than transmitting each individual pixel. The received image will also take less space when stored in the memory of the receiving fax machine or computer.

2.0 Previous work

Image processing software for the DOS operating system was written in the C language by Horst Rudolph Mellenberg in April 1992 for an independent project in for Virginia Tech. The software is well documented in Mellenberg's report [3]. That software served as the basis for this project. It also served as a valuable reference of working C code.

That software implemented several basic image processing functions in both spatial and frequency domains. It works for images with 32 gray scale levels, while images used for this project have only two: black and white. The functions of Mellenberg's software may be used to process the bit-mapped files created by this project.

For this project, Mellenberg's software was ported to the BTOS operating system. This project kept the same basic main menu of the previous software and added two new choices to process bitmapped fax image files and to process run-length encoded fax data. In addition, the bitmapped input routine now automatically reads the size of the file being input. All the original functions have been retained. Functions added for this project are: portions of Main to read fax data files (bitmapped and run-length), fax display function for video and laser printer, and four error concealment routines. The code in appendix a contains only those functions modified or created for this project. Mellenberg's functions have been appended to versions of this code not included in this report.

3.0 Display code

The display algorithms written for this project make use of a software product called BTOS Graphics II, which includes functions to display several basic geometric shapes on a video monitor. The algorithm also calls several functions supplied with the BTOS II operating system. These functions initialize and finish the video graphics display mode. There are also calls to display, track, and acknowledge input from a mouse. These mouse calls are not integral to this program, but open the possibility for future work discussed later in this report. The rest of the functions come from BTOS C, and should be compatible, for the most part, with ANSI standard C.

3.1 Reading a run-length encoded file

Although a run-length encoded image could be displayed without conversion to a bitmapped file, the conversion is done so that additional processing may be done on the bitmapped file after it is displayed. A file of data received via fax would normally be binary data consisting minimally of codewords of various lengths representing white runs, black runs, and end-of-line, according to modified Huffman coding. The data format chosen for this project is an ASCII file of integer values representing run lengths and colors. Each run length value is separated by a comma from a zero or a one representing the color of that run. Another

comma follows the color value if that run is followed by another run in the same row. The end-of-line code is represented by the carriage return/line feed character following the color code of the last run of the row. Figure 1 is a sample of this code. (The carriage return/line feed character does not show in the printed file.)

This software reads data from a run-length encoded ASCII file, converts each run to a string of ones or zeroes, and stores the strings in a new bitmapped ASCII file. The run-length reading function is called from the main function when selected from the main menu. Pointers to the run-length file and to the new bit-mapped file are passed to the function. Reading data from the run-length file, the software shifts in digits until a comma is read. It then stores the length of that run and adds it to the cumulative length of the row. The software then reads a digit representing the run color and writes this character to the bitmapped file as many times as called for by the run length. Provided an error is not detected, the next run length and color are read and written to the bitmapped file until the carriage return/line feed character is read. This character is written to the bitmapped file and the next line of runlengths and colors is read and decoded. This process continues until the end of file (EOF) is detected.

At EOF control is passed back to the main function, where the new bitmapped file becomes the input file, and the run-length file is closed.

32,0
32,0
32,0
14,0,14,1,4,0
18,0,2,1,12,0
19,0,2,1,11,0
20,0,2,1,10,0
6,0,1,1,10,0,1,1,3,0,2,1,9,0
6,0,1,1,10,0,1,1,4,0,2,1,8,0
6,0,1,1,10,0,1,1,5,0,2,1,7,0
6,0,1,1,5,0,1,1,4,0,1,1,6,0,2,1,6,0
6,0,1,1,5,0,1,1,4,0,1,1,7,0,2,1,5,0
6,0,1,1,5,0,1,1,4,0,1,1,7,0,2,1,5,0
3,0,19,1,3,0,3,1,4,0
6,0,1,1,5,0,1,1,4,0,1,1,7,0,2,1,5,0
6,0,1,1,5,0,1,1,4,0,1,1,6,0,2,1,6,0
6,0,1,1,5,0,1,1,4,0,1,1,5,0,2,1,7,0
6,0,1,1,10,0,1,1,4,0,2,1,8,0
6,0,1,1,10,0,1,1,3,0,2,1,9,0
6,0,1,1,10,0,1,1,2,0,2,1,10,0
19,0,2,1,11,0
18,0,2,1,12,0
17,0,2,1,13,0
16,0,2,1,14,0
14,0,13,1,5,0
14,0,13,1,5,0
14,0,13,1,5,0
14,0,13,1,5,0
14,0,13,1,5,0
32,0
32,0
32,0

Figure 1

Run length encoded data

3.2 Bit-mapped file display

After a run-length encoded file is translated into a bit-mapped file, the image is displayed using functions of BTOS Graphics II. If a bit-mapped file already exists, it may be read directly into this program by making the appropriate choice from the main menu.

BTOS Graphics II includes two display modes: device dependent and device independent. The device independent mode automatically scales the image to the video screen in use, but it also requires that the programmer allocate a block of memory to the displayed image. If the C code is linked with the small memory model, the amount of memory allocated for data will not be large enough to hold images much larger than 32x32 pixels. Larger images will overflow the assigned memory area and cause the workstation to hang or to crash. The small memory model makes the most efficient use of memory, and it is the model that BTOS system functions were designed for. Because of these difficulties, the device dependent mode was chosen.

The device dependent mode automatically allocates sufficient memory to display the image, unlike the devive independent mode. It does not automatically scale the image to the screen in use, but this can be accounted for with a function in the BTOS Graphics II package. The function GetVirtualPixels returns the number of actual pixels in the height and width of the screen in use. The programmer can

then use these values to scale the image, as discussed below.

Upon input, the number of characters in the first row of the file is counted to determine the width of the image. This assumes that all rows are the same length. If the image was decoded from a run-length this is assured. That error detection and concealment technique is discussed in the next section. Each pixel value read in becomes an element of the bit-mapped array.

Next the number of carriage return/line feed characters is counted to determine the length of the image. A virtual coordinate system is then set up so that the video screen is divided into small squares as if by a grid. Each of these squares is an actual pixel on the video screen. In order to scale the image to the video screen, each pixel of the image is multiplied by a scaling factor, so that the image occupies the entire screen. This scaling factor is equal to the actual number of screen pixels in the width or height of the screen divided by the either the actual number of image elements in the image height or width, whichever is greater. If an image has 64 pixels in each row and column, each pixel will occupy 1/64 of the width of the screen. The origin of this grid is the lower left corner, and each virtual pixel of this grid will correspond to a pixel value of the bit-mapped file, where the top rows or the rightmost columns may not be used if the image is not square. By choosing the

larger of the image's height or width values to scale the image, the display will not be stretched or squeezed. This also allows any size image to be displayed, within the number of actual screen pixels.

Because the screen axes are set up with row zero at the bottom of the screen, and the array row zero is the top of the image, writing each array row to the same numbered screen row would produce an upside down image. To prevent the array is read from the last row to the first and printed from the bottom screen row, row zero, to the top.

Next, each row of the bit-mapped array is scanned until a 1 character is found. The number of sequential ones is counted to determine the run length. A shaded rectangle is then drawn from the lower left corner of the virtual grid square corresponding to the first black pixel to the upper right corner of the virtual grid square corresponding to the last black pixel of the run. Shaded rectangles are drawn to correspond to each run of black pixels until the entire coded image is displayed.

The user may print this image to a laser printer. This printing function makes use of the BTOS network spooled printing utility called Generic Print Service (GPS), and the BTOS system command called Quick Screen Print. Quick Screen Print must be installed prior to running the program. Installation may be done by the user, by invoking the command Install Quick Screen Print and supplying the

printer's queue name enclosed in square brackets. The queue name is used by GPS to route the image file to the desired printer. Alternately, the Quick Screen Print installation may be placed in the system initialization file, and done automatically when the workstation is booted. Alternately, the Graphics II library also supports direct and spooled printing, but memory requirements limit this option to simpler images, generally 32x32 bits, as discussed above.

Figure 2 is the bit-mapped data created from the Figure 1 data (with bit errors added by the programmer). Figure 3 shows the laser printer output from this file. The following two pages, Figures 4 and 5, show the output from 64x64 and 128x128 pixel images, respectively.

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000111111111111110000
00090000000000000011000000000000
00000000000000000001100000000000
00000000000000000000110000000000
00000010000000000100011000000000
00090010000000000100001100000000
00090010000000000100000110000000
00090010000010000100000011000000
00000010000010000100000001100000
00000010000010000100000001100000
00011111111111111111110001110000
00000010000010000100000001100000
00000010000010000100000011000000
00000010000010000100000110000000
00000010000000000100001100000000
00000010000000000100011000000000
00000010000000000100110000000000
00000000000000000001100000000000
00000000000000000011000000000000
00000000000000000110000000000000
00000000000000001100000000000000
00000000000000111111111111100000
00000000000000111111111111100000
00009000000000111111111111100000
00000000000000111111111111100000
00000000000000111111111111100000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Figure 2

Bit-mapped data

(errors are shown as 9)

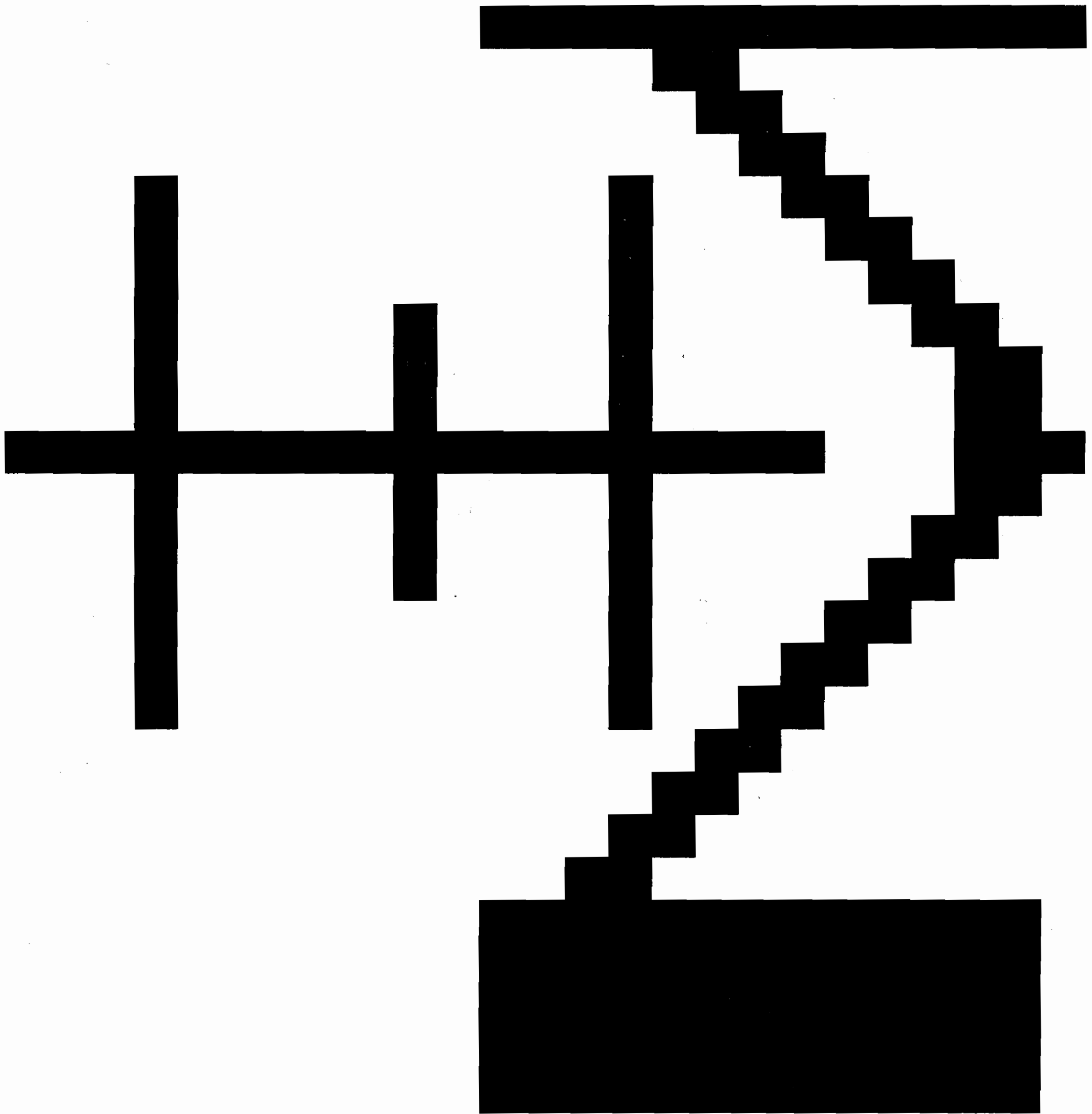


Figure 3

32x32 pixel output from Figure 2 data

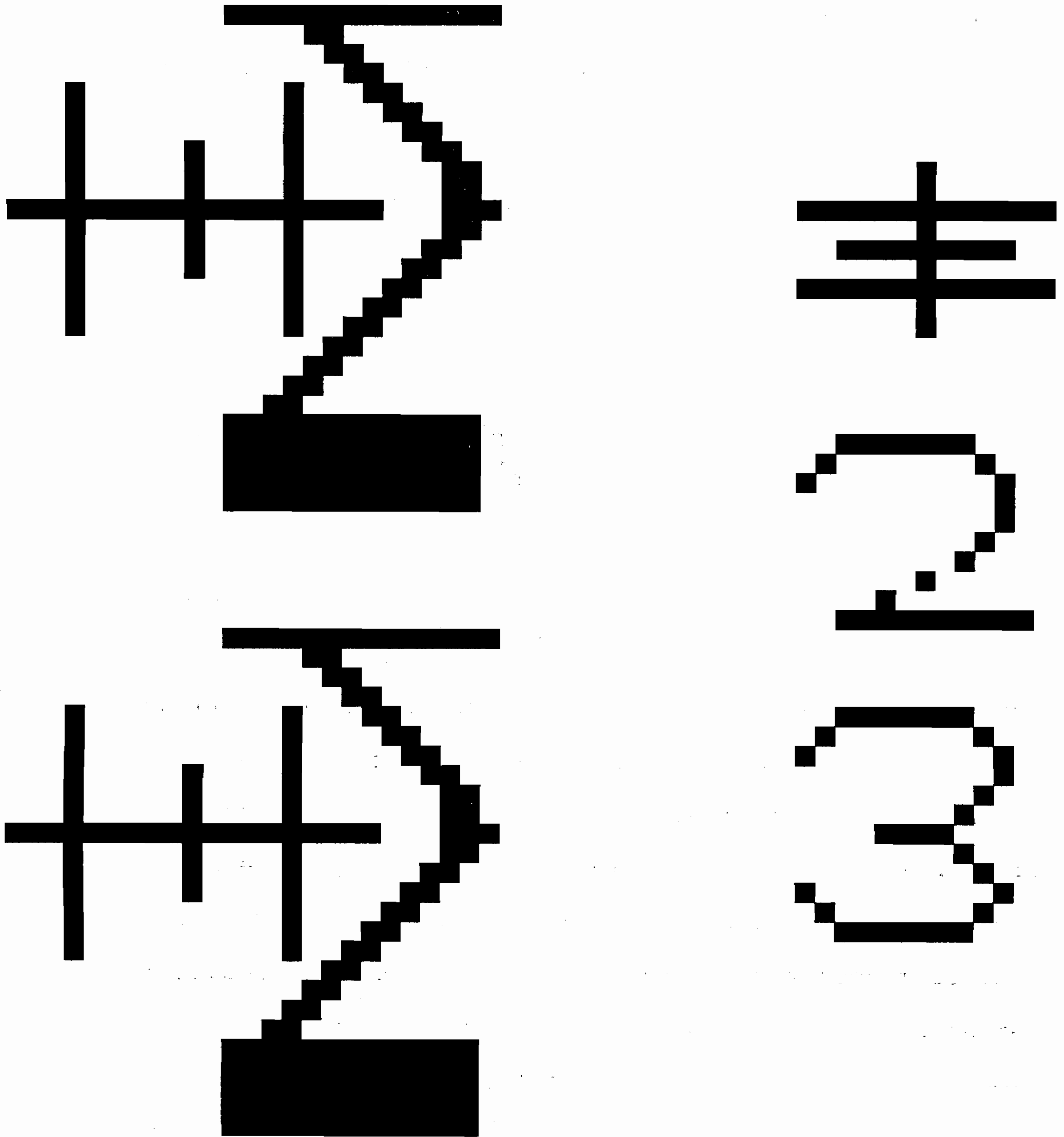


Figure 4

64x64 pixel output

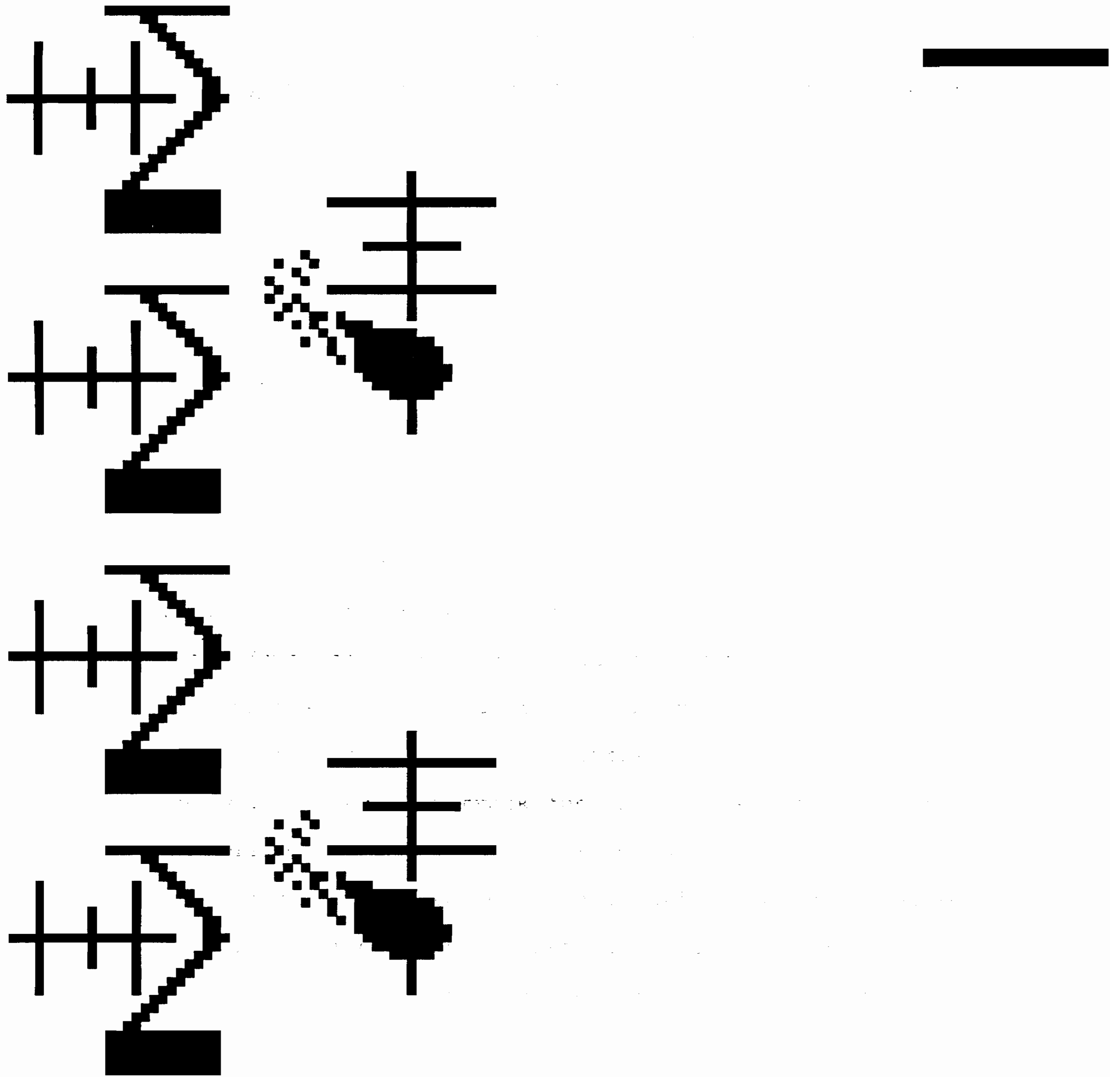


Figure 5

128x128 pixel output

4.0 Error detection and concealment

Once an image is digitized, there are a number of techniques which can be employed to improve the quality of the image or to bring out certain characteristics of the image. The four techniques applied to this project operate in the spatial domain.

When an impulse-type error occurs in the transmission of run-length encoded data, affecting just a bit or two at a time, it will look like a burst error, affecting a string of bits when the image is reproduced. This is due to the run-length encoding scheme used for fax data compression. If one bit of a code word describing a run-length is in error, the entire run will be in error. Furthermore, the total number of bits in the row will be changed, so the rest of the row will be in error. Burst errors will, of course, continue to appear as burst errors. There are many types of error correction codes and techniques which allow data to be restored to some extent if it is corrupted. Some fax machines employ a technique often called "modified modified read", which encodes differences between the row being recorded and the following row [2]. This technique is effective because fax data often contains a large degree of vertical correlation. Error correction encoding is not considered in this project, but the high degree of vertical correlation often found in fax images does make the four error concealment techniques used in this project effective.

4.1 Error detection

When run-length data is read into this program, the user tells the program how many bits should be in each row of data, and the program adds each run to the cumulative length of each row before it writes that run to the bit-mapped file. If the new length would be longer than the allowed length, there is an error. The program also checks for end of line code before the allowed length of each row is reached; the absence of a comma following the color code digit; and EOF indication before an entire row is read.

All of these errors invoke the same error function. The calling function passes to the error function the bit-mapped file pointer, and the number of bits successfully written to the present row of the bit-mapped file. The error function writes the 9 character to the bit-mapped file to fill the rest of the bits of the present row of the bit-mapped file, then writes the carriage return/line feed character. Next the track function reads characters from the run-length file until it finds an end of line. Then it returns control to the calling function, which begins to read a new line. If the error function reads EOF, it gracefully ends the routine and allows the program to continue. The use of the end of line character allows errors in one line of data to be confined to that row.

When the bitmapped file is printed, only strings of black pixels, the 1 character, are shown. This means errors, any other character than 1, show up as white space, rather than some other strange character. This helps minimize the negative impact of unknown characters, while still preserving the fact that there is an error in the stored bit-mapped file.

4.2 Error concealment

This software emulates four error concealment techniques commonly found in fax machines, as described in McConnell, Bodson, and Schaphorst [2]. The Print White (PW) technique prints the first line with an error as white. All subsequent rows are white until an error-free row is found. Then normal display continues until the next error.

The Print Previous Line (PPL) technique prints the last error-free row in place of a row with an error. The last error-free row replaces all subsequent rows until another error-free row is detected.

The previous two techniques may also be combined by replacing the first line with an error with the previous line, but replacing all subsequent rows with errors with white rows until another error free row is found. This is known as Print Previous Line/White (PLW).

Finally, Normal Decode/Previous Line (NDPL) prints all correct pixels in a row containing an error up to the first error. For the rest of that row, the corresponding pixels

from the previous row are printed. This corrected row is now considered "correct", and the same technique is applied to the next row, if it also contains an error. Figures 6-9 show the data in Figure 2 printed using each of these four error concealment techniques.

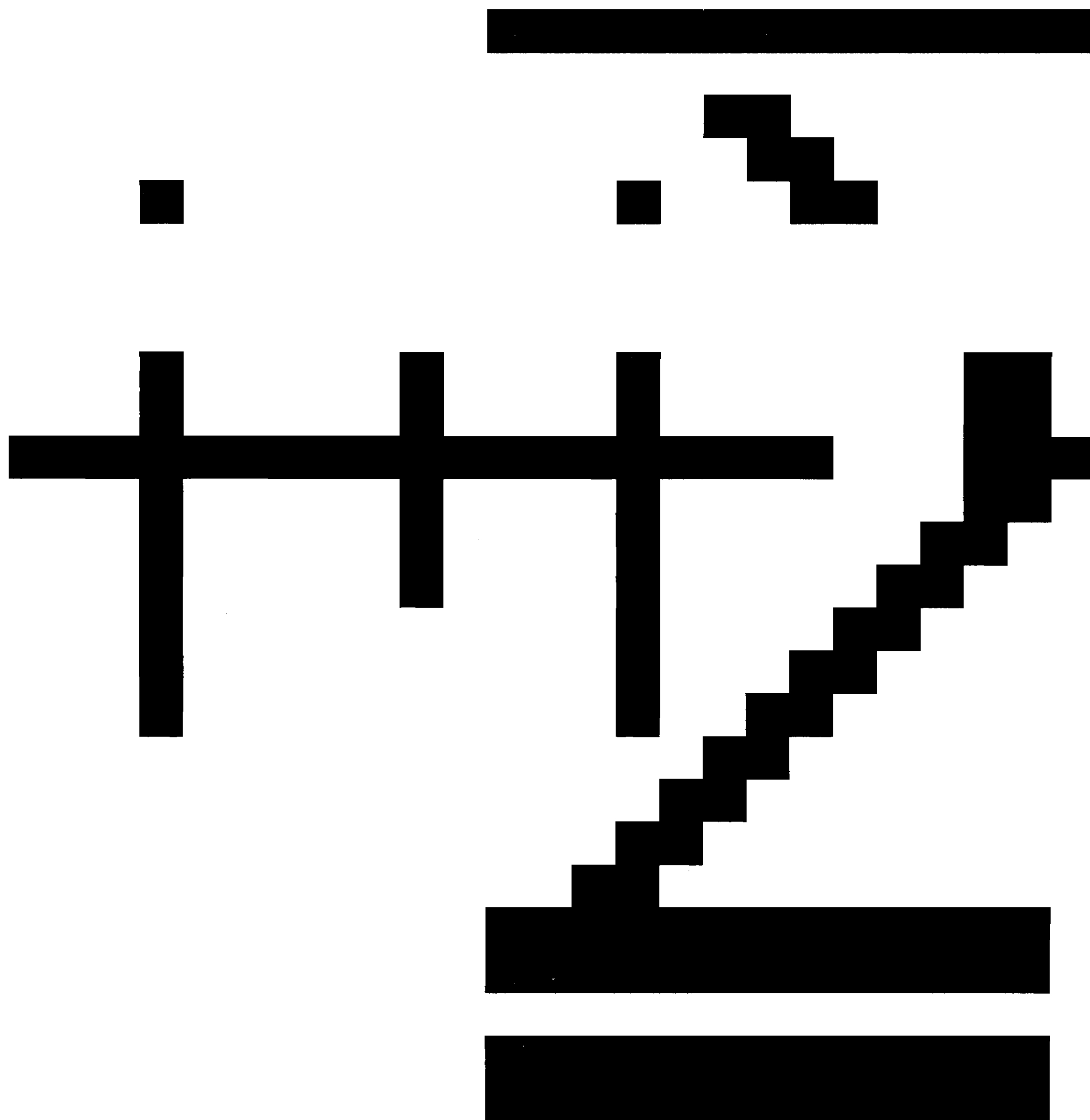


Figure 6

PW error concealment

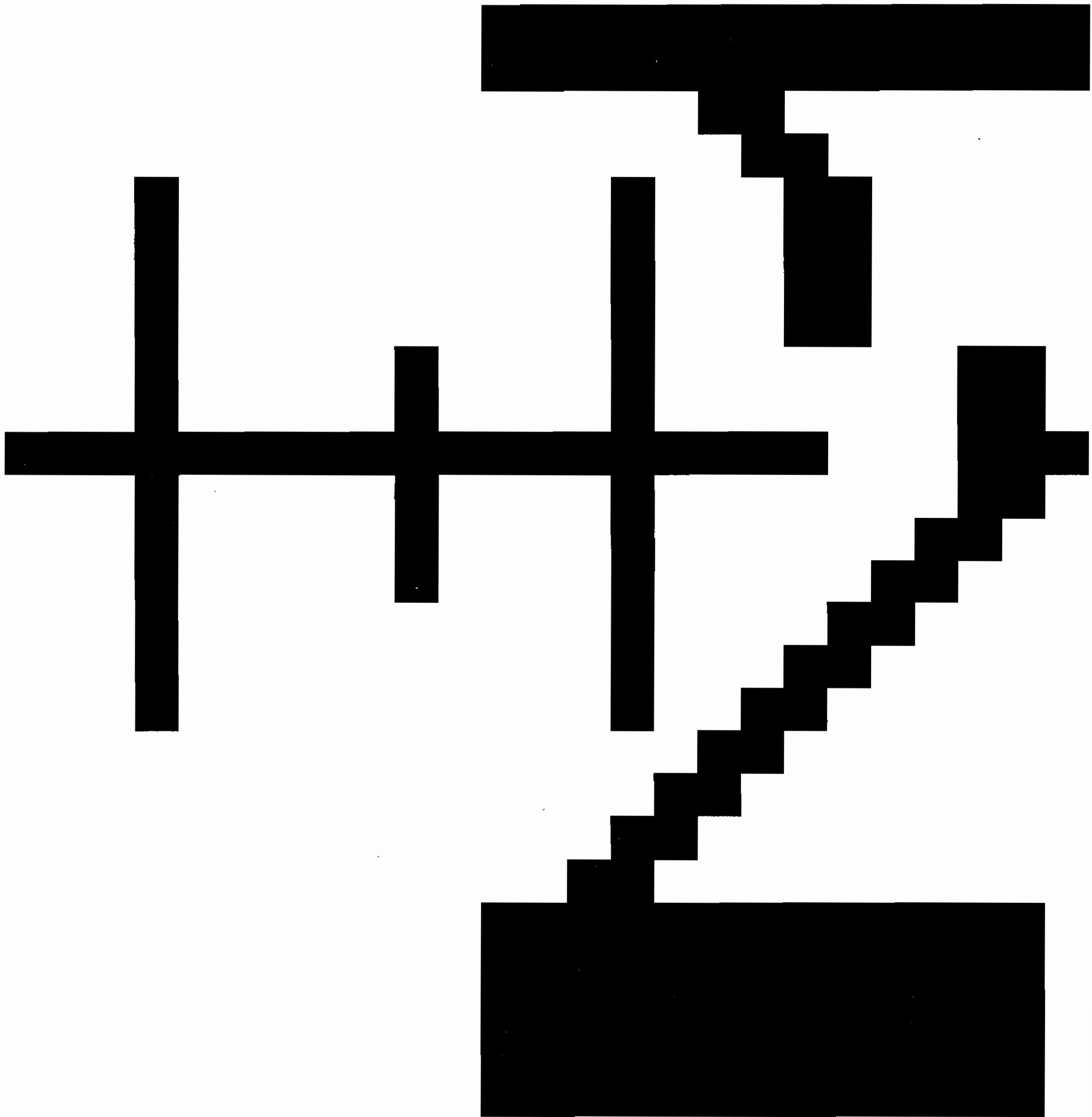


Figure 7

PPL error concealment

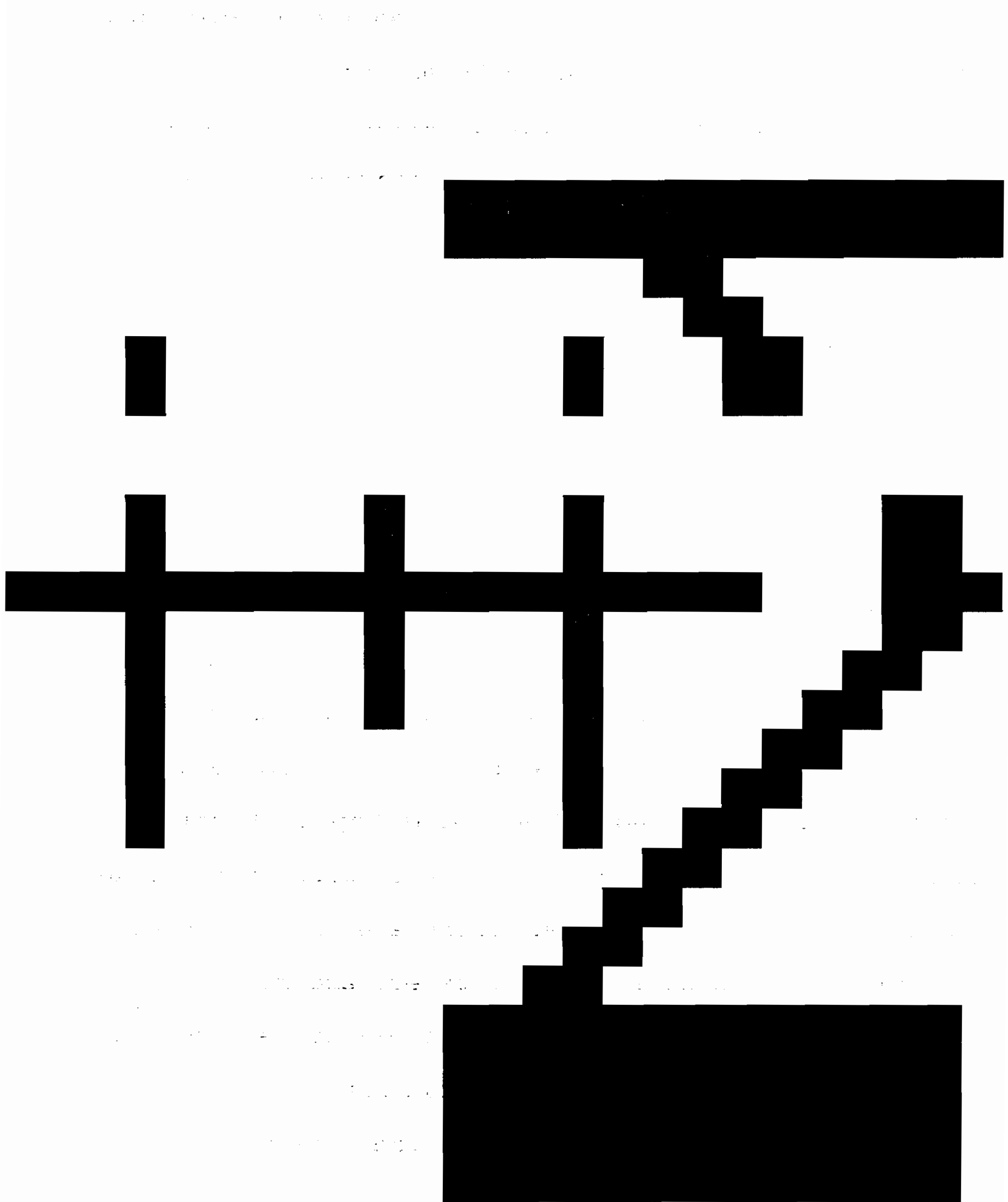


Figure 8

PLW error concealment

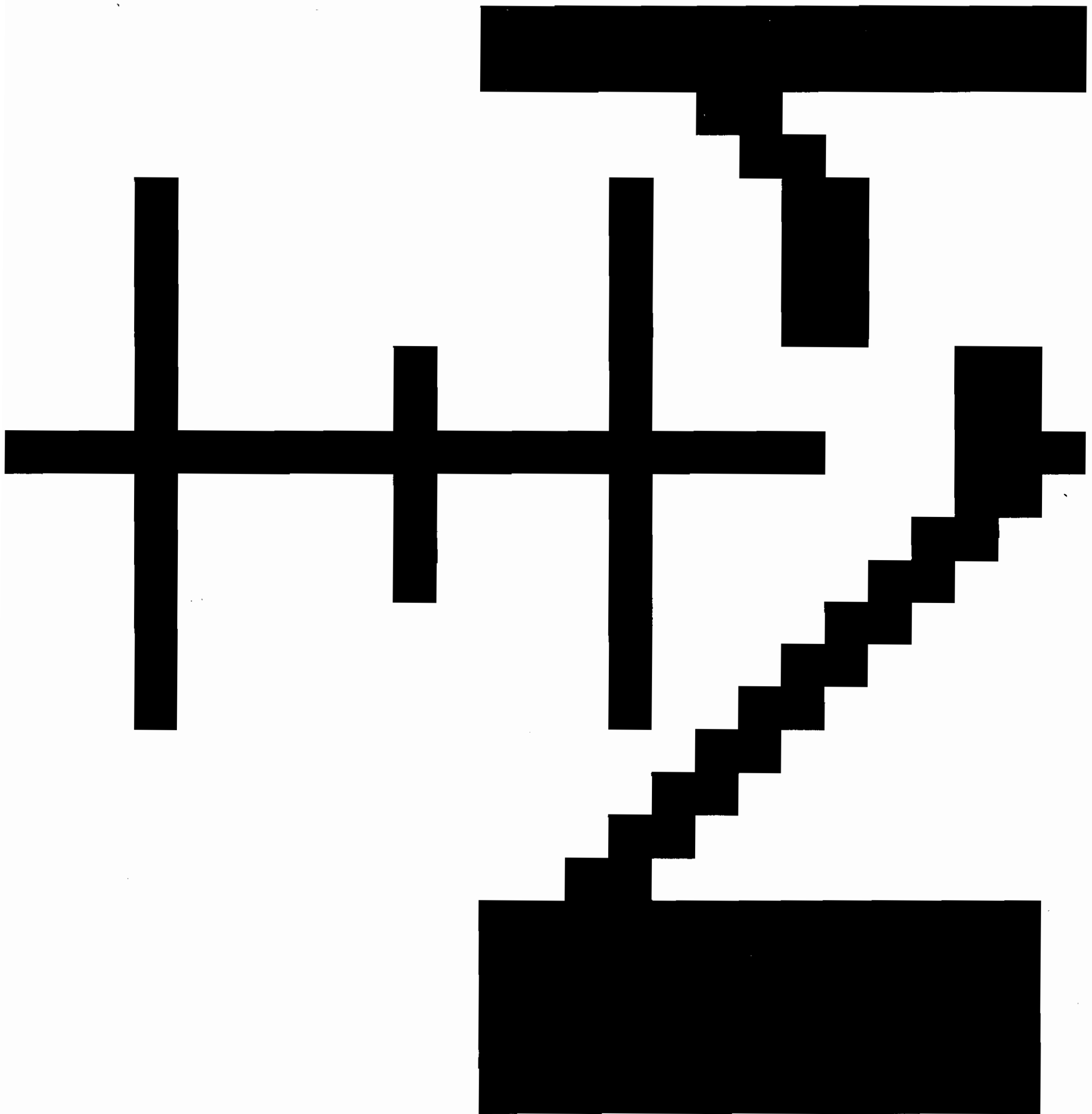


Figure 9

NDPL error concealment

5.0 BTOS interfaces

Working with the BTOS operating system posed several challenges, primarily because it is not an industry standard. The first challenge was to find software that would allow a programmer to draw the lines or filled rectangles to the display screen. The documentation for the BTOS Graphics II package was located on a bookshelf, without the software diskette. A marketing representative for the Unisys Corporation, which sells the software, was unaware of its existence or whether a license had already been purchased in the past, much less anyone who could explain its capabilities and known limitations. Fortunately, the software diskette was located, languishing in an unmarked binder with other unrelated software. The documentation for this product proved to be well written and helpful [4].

The BTOS system documentation, on the other hand, proved difficult to use for anyone not already familiar with programming in this environment [5]-[6]. Descriptions of system functions are sparse, and lacked instructions for calling the functions with various programming languages. The function arguments for this project were declared primarily using intuition, trial, and error.

When linking the compiled code, in addition to the object file for the project code, the object file c0s.obj must also be included. Also, this project required including the following libraries when the object files are linked: clibs.lib, maths.lib, graphics.lib, GPAM.lib and mouse.lib.

6.0 Project accomplishments

Perhaps the biggest accomplishment of this project was finding and developing software to print a bit-mapped file to the screen of a Coast Guard Standard Workstation and to a laser printer. This produces a smoother appearing one bit output than the previous technique using the overstrike capability of a dot matrix printer, and also allows the image to be viewed on a video screen.

Software was also developed to read a modified run-length encoded file and write it as a bit-mapped file. The encoding of the relatively straightforward error concealment techniques also showed promise as effective error correction mechanisms.

An important by-product of this project has been gaining proficiency in the C programming language. While admittedly this project has not created an expert, it has provided valuable experience with the language where none existed before.

Programming on the Coast Guard Standard Workstation has proven to be a challenge. From difficulty in identifying, procuring, and implementing necessary software, to deciphering difficult documentation, these results have not come easily. These tribulations, however, should prove valuable in future endeavors with this platform.

7.0 Future work

The algorithm used by this software is limited by the BTOS C compiler to images no larger than 128x128 pixels, because the entire image is read into a single array for display and processing. BTOS C limits an array to no larger than 128x128 elements. Future work on this project could explore ways to read only part of an image at a time into the array, yet display a large image by reading data directly from the bit-mapped or run-length file.

The user interface would also be greatly enhanced with full implementation of the mouse. This could allow the user to select menu choices without using the keyboard. It could also allow the user to choose small portions of a large image for processing by marking and bounding rectangles on the displayed image using the mouse. This may allow a 128x128 pixel portions of a large image to be processed. It would also allow users to choose only the corrupted part of the image for processing.

An obvious next step would also be to read streams of actual bits of modified Huffman code, and also decoding error correction codes.

Also, the display algorithms could be modified to display more than two gray scale levels by using different types of rectangle shadings and patterns. BTOS Graphics II offers ten different rectangle shadings, and it may be possible to develop others to represent a greater range of gray scales.

References

[1] P. Wintz and R. C. Gonzalez, *Digital Image Processing*. Addison-Wesley, Reading, MA, 1987.

[2] D. Bodson, R.Schaphorst, and K.McConnell, *FAX: Digital Facsimile Technology and Applications*. Artech House, Boston, 1992.

[3] H. R. Mellenberg, "An Evaluation of Picture Processing Techniques." Independent Study Report for Virginia Polytechnic Institute and State University, April 1992.

[4] Unisys Corporation, *BTOS Graphics II*. Unisys Corporation, 1987.

[5] Unisys Corporation, *BTOS II System Reference Manual*. Unisys Corporation, 1988.

[6] Unisys Corporation, *BTOS II System Procedural Interface*. Unisys Corporation, 1988.

[7] M. Waite and S. Prata, *C: Step-by-Step*. Prentice Hall, Carmel, IN, 1992.

Appendix A

Program Listing

```
/*************************************
 *            RUN11.C                *
 *                                   *
 *       BY PATRICK H. GEDDES        *
 *                                   *
 *                                   *
 *                                   *
 *          07 December 1992         *
 *************************************/
/*************************************
 *                                   *
 *     THIS PROGRAM PROCESSES        *
 * a) RUN-LENGTH ENCODED IMAGES      *
 * b) BIT-MAPPED BLACK AND WHITE IMAGES*
 *                                   *
 *  INPUT: PICTURE (128 by 128 PIXELS) *
 *  REPRESENTED BY ASCII CHARACTERS  *
 *  (0 and 1)                        *
 *                                   *
 *  OUTPUT: PROCESSED PRINTED PICTURES *
 *
 *  PRINTED COPY BY USING QUICK      *

 *         SCREEN PRINT              *
 *************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#define MAX(a,b)    (((a) > (b)) ? (a) : (b))
#define MIN(a,b)    (((a) < (b)) ? (a) : (b))
#define ROUND(a)  (((a) < 0) ? (int)((a)-0.5) :
(int)((a)+0.5))

char   Runcode[13];
char   infilename[81], bitfilename[81],
              int_array[64][64], char_pixel;
char   big_array[128][128];
int    law, neg, nx, ny, i, j, menu;
       int    siz[2];
       FILE   *infile, *outfile, *temp;


char *get_string(char *);
int get_int(char *,int,int);
```

```
  double get_float(char *,double,double);

            void FaxDisp(char [128][128], int [2]);
                  /* Displays bitmapped black and white image
*/
            void Conceal(char [128][128], int[2]);
                  /* Fax Error concealment routines */
            void Runlength(FILE *, FILE *);
                 /* Translates run-length encoded file
                    to bit-mapped format */
            void correct(FILE *, int [2]);
                 /* Accounts for errors reading
                    run-length encoded file */
            int  track (FILE *);
                 /* Gets reading run-length data routine
                    back on track after an error */

void Error (int erc) {
   if (erc !=0) ErrorExit (erc);
}




  main()
  {

char  pixel;
void *malloc(int);
int   fun;

  printf("\n                        A Picture Processing
Program\n\n");
  printf("\n");
  printf("Enter type of image processing:\n");
  printf("    <x> Bit-mapped fax routines\n");
  printf("    <y> Read run-length encoded fax file\n");
  printf("    <q> Quit\n");
  printf(">>: ");


  if (((menu=getchar())>='a' && menu<='l') || menu=='q' ||
                menu=='x' || menu=='y') {
      putchar('\n');
  }
  else printf("\n\nERROR: You did not enter lower case a - l
or q...\n\n");
  if (menu=='q') exit(0);
         if(menu=='l')  goto end;
  printf("\nEnter Input Filename: ");
```

```
  scanf("%80s",infilename);

      if((infile = fopen(infilename,"r"))==(FILE *) NULL ) {
   printf("Couldn't open %s for writing.\n",infilename);
   exit(1);
  }
/*******************************************************/
/*  Read run-length encoded data from input file      */


if (menu=='y') {
         printf("\nEnter output filename for bitmapped
file\n");
  scanf("%80s",bitfilename);

  if((outfile = fopen(bitfilename,"w+"))==(FILE *) NULL ) {
      printf("Couldn't open %s for writing.\n",bitfilename);
      exit(1);
  }
/*  Call routine to translate run-length file    */

        Runlength(infile, outfile);

/* Output of this routine becomes the input for following
routines  */

        temp=infile;
        infile=outfile;

      fflush(temp);
      fclose(temp);
      rewind(infile);
/* data is now in bit-mapped format        */
/* chain to bit-mapped fax routines        */

      menu= 'x';
}

/*  Read bit-mapped input file into an array      */
/*  Count number of characters in first input row*/
/*  This becomes number of filled array columns   */

while (pixel != '\n'){
        pixel = getc(infile);
        nx++;

 }
nx -= 1;

rewind(infile);
```

```
/*  Count number of rows of data                        */
/*  This becomes number of filled array rows            */

while ((pixel = getc(infile)) != EOF) {
      if (pixel == '\n') ny++;
}

/*  array siz[] is used to pass both
    nx and ny to functions with a single pointer */
siz[0] = nx;
siz[1] = ny;

rewind(infile);
fflush(infile);

printf("\n %d Rows and %d Columns in input file\n", ny, nx);

/* Read bit-mapped fax data into character array */

if (menu == 'x')
{

for (i=0; i<=ny; i++)  {
    for (j=0; j<=nx; j++)  {
        pixel = getc(infile);
        if (pixel != '\n') {
        big_array[i][j] = pixel;
        }
    }
 }
fflush(infile);
fclose(infile);

/*  label for jump if q selected from menu             */
    end:
```

```
  switch (menu) {

/*  display bit-mapped image and call error concealment
routines  */

  case 'x':  FaxDisp(big_array,siz);
                   printf("\nEnter 1 to try error
concealment techniques,\n");
                   printf("Enter anything else if you
don't.\n");
                    if ((getchar()) == '1'){
                        Conceal(big_array,siz);
                    }
                   break;
  case 'q': printf("\n\n"); return 0;
  default: printf("\nError: menu failure\n"); getchar();
return 0;
  }

  fflush(outfile);
  fflush(infile);
  fclose(infile);
  fclose(outfile);

      }


}

/***********************************************/

/*          Fax: Decode run length encoded file */

/*         INPUT: infile (run length encoded)   */
/*         OUTPUT: new infile (bitmapped)       */

/*         P. H. GEDDES
/***********************************************/

void Runlength(FILE *runfile, FILE *bitfile)


{
char digit;
int colored, two, m, last, newline, Howlong, length[2], run,
color;

printf("\nEnter number of characters in a line of input
file\n");
scanf("%d", &two);
```

```
/* Last is switch enabled at EOF   */
last=0;

length[1]=two;

/* Continue reading characters until EOF is reached */

while (last == 0){

   length[0]=0;
   run=0;
   newline = 0;
   colored = 0;


/* newline flag is set when end of line  */
/* character is read                     */
/* Howlong is a switch set when          */
/* "how long a run is" is known because  */
/* the separating comma is read          */
/* length[0] keeps track of how many     */
/* pixels are in the row so far          */

   while(newline ==0){         /*line length test  */
     run=0;
     Howlong = 0;
     while (Howlong == 0){
        digit=getc(runfile);
        switch (digit)
        {
        case '\n': if(length[0] != 0){
                        correct(bitfile, length);
                        fprintf(bitfile,"\n");
                        if(track(runfile) == 1){
                              return;}
                        length[0]=run=0;
                   }

                   break;

        case EOF:  if(length[0] !=0 || length[0]<length[1]){
                   correct(bitfile, length);
                   }
                   fprintf(bitfile,"\n");
                   return;

        case ',':  if (colored == 1) {
                      colored=0;
                   }
                   else Howlong=1;
```

```
                  break;

         default :  run = (run*10) + (int)digit - (int)'0';
                    colored=0;
                    break;

         }                    /* end switch  */
      }                      /* while Howlong ==0 */

/* colored is a switch set when      */
/* a run length is read              */
/* it signals what follows the next */
/* comma is the beginning of         */
/* the next run length.  If colored */
/* is not set, what follows the      */
/* comma is a color value.           */

colored=1;
color=getc(runfile);

         switch (color){

         case '\n': correct(bitfile, length);
                    if(track(runfile) == 1) return;
                    newline=1;
                    break;

         case EOF: correct(bitfile, length);
                   return;

         case ',': correct(bitfile, length);
                    if(track(runfile) == 1) return;
                   newline=1;
                   break;

         default : if (length[0] + run>length[1]){
                      correct(bitfile, length);
                          if(track(runfile) == 1){
                               return;}
                      newline=1;
                   }               /*  if  length[0]...   */
                   else{
                       length[0] += run;
                       for(m=0; m<run; m++){
                           fprintf(bitfile,"%c",color);
                          }
                       if (length[0]==length[1]){
                          newline=1;
                          }
                    }                        /* else   */
                   break;
```

```
        }                    /* end switch  */

    }       /*line length test: while newline==0  */

fprintf(bitfile,"\n");
}       /* while last = 0*/

return;
}

/************************************************************
*/

/*            Fax: Correct errors in run length encoded file
*/

/*           INPUT: new infile (bitmapped), length
*/
/*           OUTPUT:
*/
/*           P. H. GEDDES
*/

/************************************************************
*/

void Correct(FILE *fp, int sofar[2])
{
int j, newlen;


/* When an input error is detected in runlength data, */
/* This routine fills in the rest of the row with 9's */
/* It is then followed by the track routine           */


newlen=sofar[1]-sofar[0];
for(j=0;j<newlen;j++){
   fprintf(fp,"9");
   }

}
```

```
/*************************************************************/

/*            Fax: get input back on track                  */

/*          INPUT: run length encoded file                  */
/*          OUTPUT: flag indicating EOF                     */

/*          P. H. GEDDES                                    */
/*************************************************************/


/* This routine gets the run-length reading back on track */
/* It looks for the next end of line character, so that   */
/* when control is passed back to the reading function, it*/
/* starts reading at the start of the next line.  If it    */
/* finds EOF, it returns a 1 value.                        */

int track(FILE *fr)
{
char nex;
int final;

final=0;
 while((nex=getc(fr))!= '\n'){
    if(nex==EOF){
       final=1;
       return final;
    }
 }


}
```

```
/************************************************************/

/*          Fax: Display and print bit-mapped image        */

/*          INPUT:  image array, number of rows, columns  */
/*          OUTPUT:                                       */

/*          P. H. GEDDES                                  */
/************************************************************/


void FaxDisp(char bitfax[128][128], int size[2])

{
typedef struct {
int wEventType;
char bCodeRet;
int wX;
int wY;
} EVENTBLOCK;

EVENTBLOCK *place;

/* functions of Graphics II and of operating system     */

int AddObject plm far (char *far, int, float, float, float,
float);
int AllocMemorySL plm far (int, char *far *far);
int AllocAreaSL plm far (int, char *far *far);
int ShrinkAreaSL plm far (char *far *far, int);
int DeallocMemorySL plm far (char *far *far, int);
/*   *far mem;   */
int CloseObject plm far (void);
int ClosePicture plm far (char);
int FillRectangle plm far (float, float, float, float,
char);
int InitGraphics plm far (void);
int Move plm far (float, float);
int OpenPicture plm far (char *far, int, char *far, int,
int, char *far, int);
int SetScreenVidAttr plm far (int, char);
int SetDrawingMode plm far (int);

int SetPlotterDevice plm far (char *far, int);
int SetOutputType plm far (int);
int SetOutputDevice plm far (int);
int DisplayPicture plm far (int);
int PDLoadCursor plm far (int *far, char *far, int);

int PDSetCursorType plm far (int);
int PDSetCursorDisplay plm far(char *far);
```

```
int ReadInputEventNSC plm far (char, EVENTBLOCK *far, int);
int PDSetTracking plm far (char *far);
int SetVirtualCoordinates plm far (char, char, char *far,
char *far);
int ClearVectors plm far (void);
int ClearViewport plm far (void);

/* void SetStyleRam plm far (int);  */
int InitScreenGraphics plm far (void);
int GetVirtualPixels plm far (int *far, int *far);
int DrawVirtualLine plm far (int, int, int, int);
int FillScreenRectangle plm far (int, int, int, int, char);
/* void ErrorExit plm far (int);  */
int ClearScreen plm far (void);
int TurnOffGraphics plm far (void);


/*  Variables */

int m, k;
int line, start;

int nx, ny;
     int VPixelsHeight;
     int VPixelsWidth;
int xscale, yscale;

/* void Error (int erc) {
   if (erc !=0) ErrorExit (erc);
} */


/* drawing part  */

printf("\nDo you want a hard copy of your image?\n");
printf("Press ACTION-COPY while image displayed");
nx = size[0];
ny = size[1];
nx = ny = MAX(nx, ny);
/*  Initialize graphics display mode                    */

     Error (InitScreenGraphics());

/*   Turn off screen refresh    */
     Error (SetScreenVidAttr (1,0));

/*   Find out actual pixels in screen being used   */
     Error (GetVirtualPixels (&VPixelsWidth,
&VPixelsHeight));
```

```
/*   compute scaling factors: screen size/image size  */
     xscale = VPixelsWidth/nx;
     yscale = VPixelsHeight/ny;

/*   count backwards so image isn't upside down      */
for (m=ny-1; m>=0; m--)  {

    for (k=0, line=0; k<=nx-1;k++)  {
        if (bitfax[m][k] == '1' && line == 0) {
            start = k;
            line = 1;
        }
        if (bitfax[m][k] != '1' && line == 1)
 {
          Error (FillScreenRectangle (start*xscale, (ny-1-
m)*yscale,
 k*xscale, (ny-m)*yscale, 0));

             line = 0;
        }

     }

}
/* Turns on mouse and holds image until mouse is clicked */

Error (PDSetCursorType(1));
Error (PDSetCursorDisplay("true"));
Error (PDSetTracking("true"));
Error (ReadInputEventNSC (0, place, 7));


/* close graphics                    */

Error (ClearScreen());
Error (TurnOffGraphics());

Error (SetScreenVidAttr (1,1));


}
```

```
/*****************************************************/

/*      Fax Error Concealment                          */

/*      INPUT: bit-mapped array                        */
/*      OUTPUT: screen display of image                */

/*      P. H. GEDDES                                   */
/*****************************************************/

void Conceal(char bits[128][128], int big[2])
{

int nx, ny, oops, count, k;
char menu;

nx = big[0];
ny = big[1];

printf("\nChoose type of error concealment:\n");
printf("a  for Print White - PW\n");
printf("b  for Print Previous Line - PPL\n");
printf("c  for Print Previous Line/White - PLW\n");
printf("d  for Normal Decode/Previous Line - NDPL\n");
printf("a=PW, b=PPL, c=PLW, d=NDPL >>");
  if (((menu=getchar())>='a' && menu<='d') ) {
      putchar(menu);
      putchar('\n');
  }
  else printf("\n\nERROR: You did not enter lower case a -
d...\n\n");

/*********/
/*  PW error concealment  */
/********/

if (menu =='a'){
   for(i=0; i<=ny-1; i++){
       for(j=0, oops=0; j<=nx-1; j++){
          if (bits[i][j] > '1'|| bits[i][j] < '0') oops = 1;

       }
     if (oops == 1){
      for (k=0; k<=nx-1; k++){
          bits[i][k] = '0';
      }
     }
   }

}
```

```
/*********/
/*  PPL error concealment  */
/********/

if (menu =='b'){

count = 0;
   for(i=1; i<=ny-1; i++){
       for(j=0, oops=0; j<=nx-1; j++){
         if (bits[i][j] > '1' || bits[i][j] < '0') oops = 1;

       }
     if (oops == 1){
      count++;
      for (k=0; k<=nx-1; k++){
          bits[i][k] = bits[i-count][k];
      }
     }
     else count = 0;
   }
}

/*********/
/*  PLW error concealment  */
/********/

if (menu =='c'){

count = 0;
   for(i=1; i<=ny-1; i++){
       for(j=0, oops=0; j<=nx-1; j++){
          if (bits[i][j] > '1' || bits[i][j] < '0') oops =
1;

       }
     if (oops == 1){
      count++;
        if (count ==1){
             for (k=0; k<=nx-1; k++){
               bits[i][k] = bits[i-1][k];
             }
        }
        else if (count >=1){
             for (k=0; k<=nx-1; k++){
               bits[i][k] = 0;
             }
        }
     }
     else count = 0;
   }
}
```

```
/**********/
/*  NDPL error concealment  */
/*********/

if (menu =='d'){

   for(i=1; i<=ny-1; i++){
       for(j=0, oops=0; j<=nx-1; j++){
          if (bits[i][j] > '1' || bits[i][j] < '0'){
              oops = j;
              j=nx;
          }

       }
     if (oops > 0){
      for (k=oops; k<=nx-1; k++){
          bits[i][k] = bits[i-1][k];
      }
     }
   }
}


/*   call display function     */
FaxDisp(bits, big);

}
```

Vita

Patrick H. Geddes received his bachelor of science degree in electrical engineering from the United States Coast Guard Academy in 1986, and was commissioned as an Ensign in the United States Coast Guard. From 1986 until 1988 he served as Deck Watch Officer aboard the U.S. Coast Guard Cutter Blackhaw, homeported in San Francisco, California. From 1988 until the present he has been stationed at U.S. Coast Guard Headquarters in Washington, D.C. From 1988 until 1990 he was the Search and Rescue Database manager. He is presently a staff officer in the Radionavigation Division, Loran-C Branch, assisting with the Coast Guard's withdrawal from overseas Loran-C operations. He currently holds the rank of Lieutenant.

Patrick H. Geddes